Pointers and Memory Allocation

CISC 3130 Notes

Michael Lampis

mlampis@cs.ntua.gr

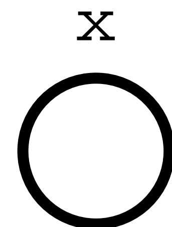
Brooklyn College

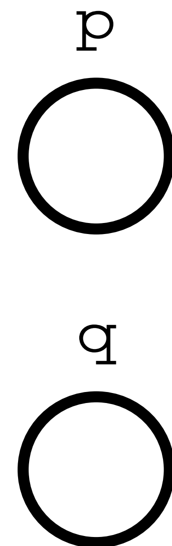
Pointers

```
int x;
```

x

Pointers



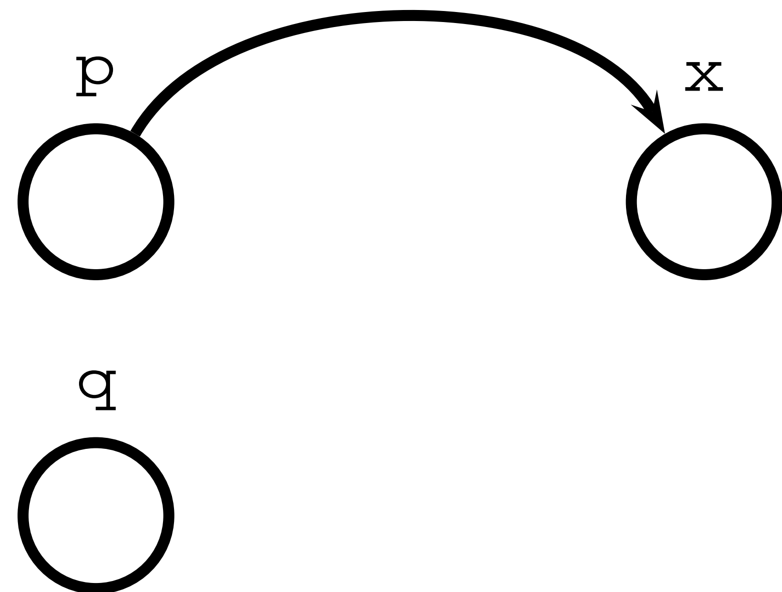



```
int x;
 int *p, *q;
```

Note:Not the same as `int *p,q;`

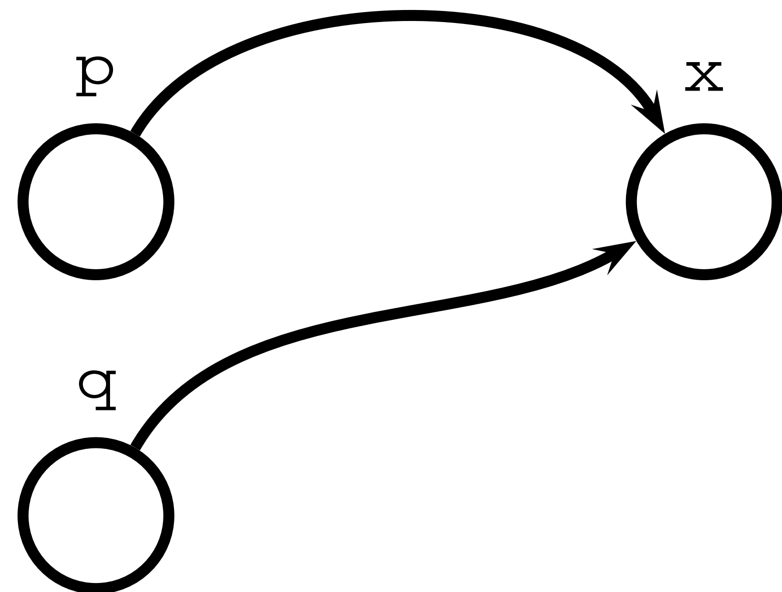
Pointers

```
int x;
int *p, *q;
p = &x;
```



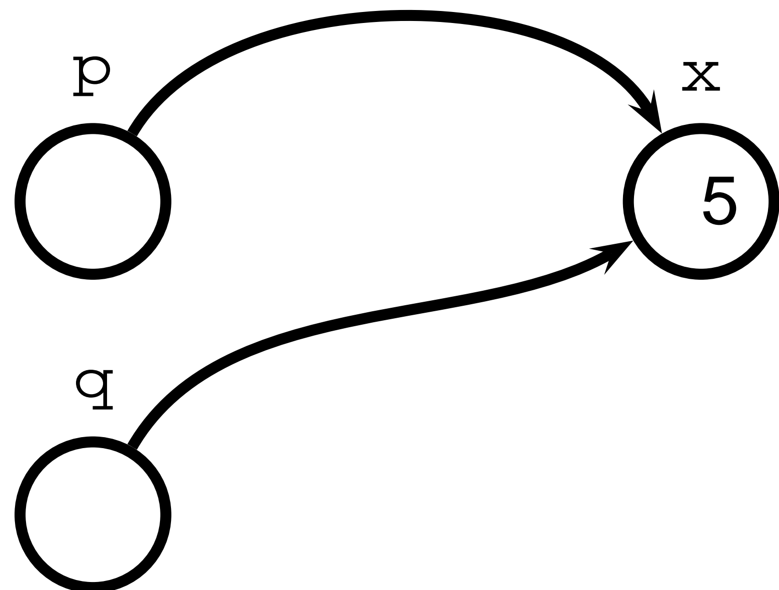
Pointers

```
int x;
 int *p, *q;
 p = &x;
 q = p;
```



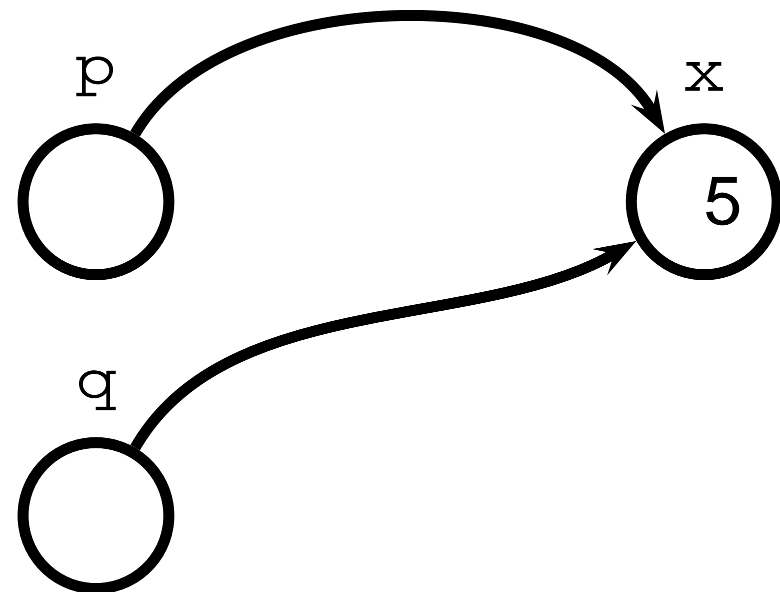
Pointers

```
int x;
int *p, *q;
p = &x;
q = p;
x = 5;
```



Pointers

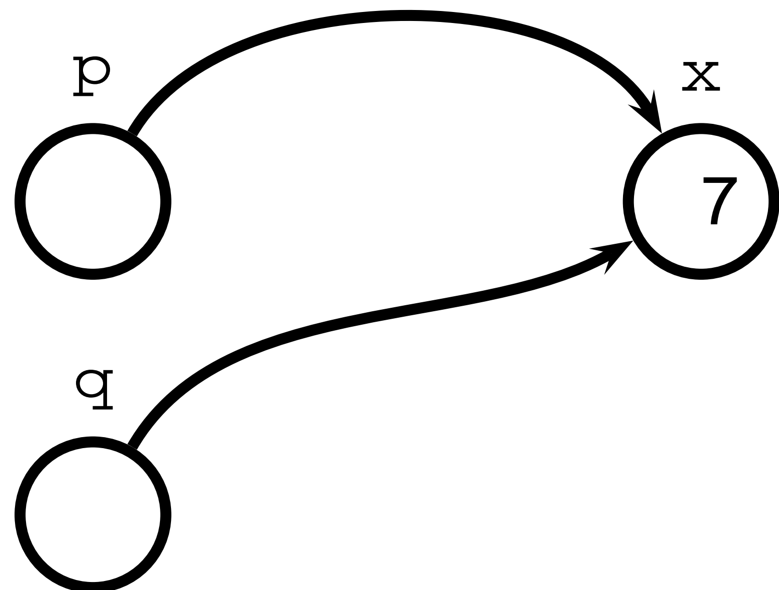
```
int x;
 int *p, *q;
 p = &x;
 q = p;
 x = 5;
```



Note:Now `*p == *q == 5`

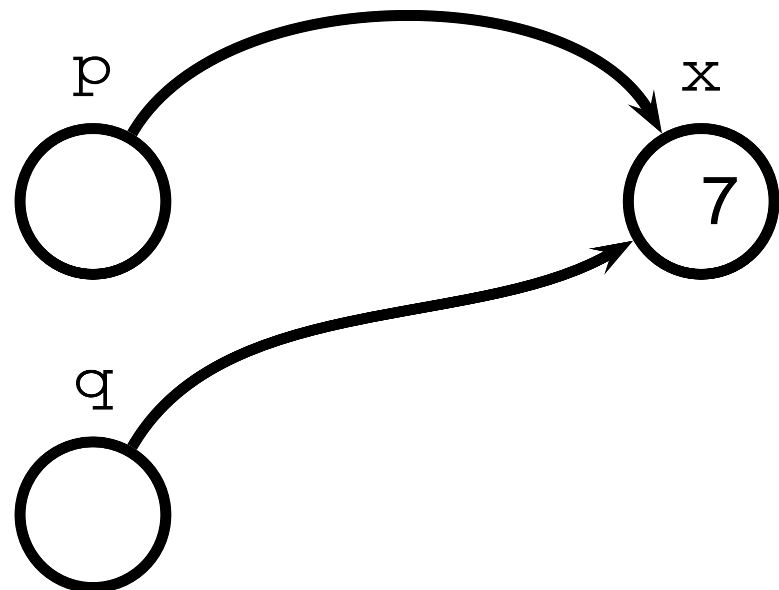
Pointers

```
int x;
 int *p, *q;
 p = &x;
 q = p;
 x = 5;
 *p = 7;
```



Pointers

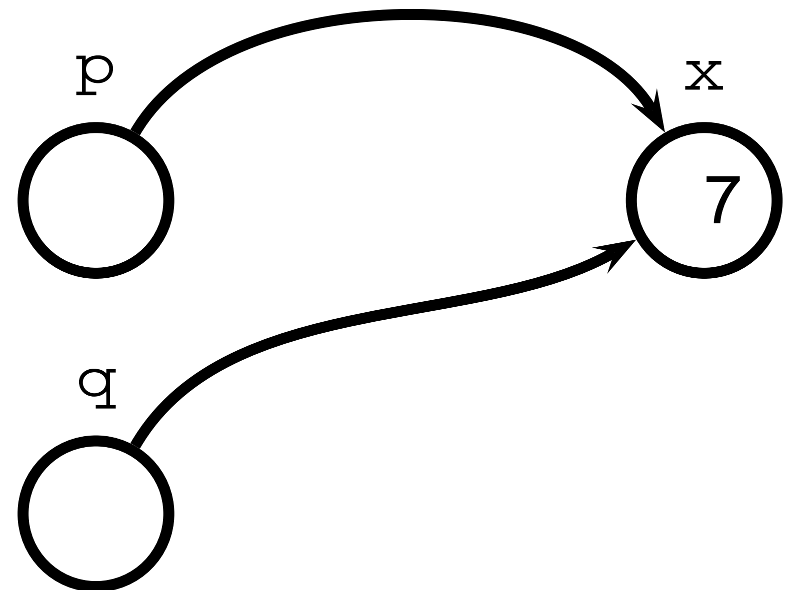
```
int x;
 int *p, *q;
 p = &x;
 q = p;
 x = 5;
 *p = 7;
```



Note: Now `x == *q == 7`

Pointers

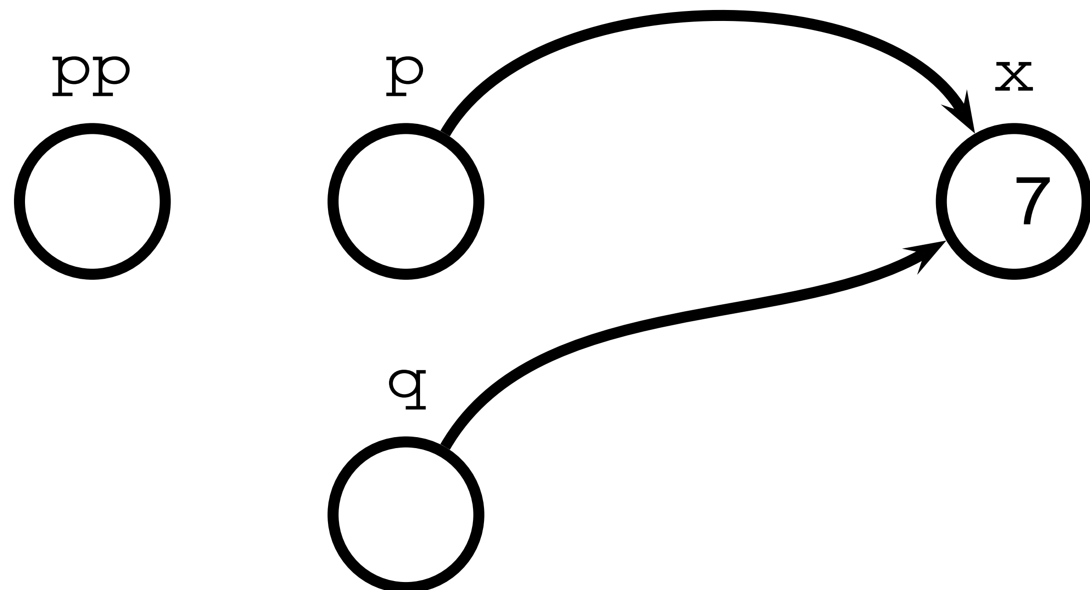
```
int x;
int *p, *q;
p = &x;
q = p;
x = 5;
*p = 7;
```



What is the difference between `p = q;` and `*p = *q ;` ?

Pointers

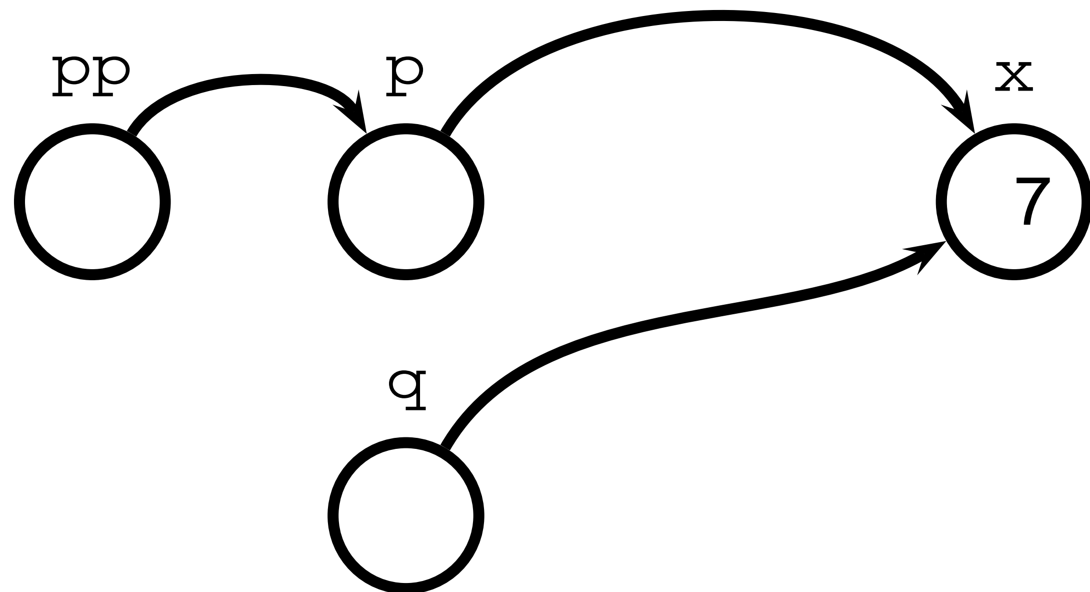
```
int x;
 int *p, *q;
 p = &x;
 q = p;
 x = 5;
 *p = 7;
 int **pp;
```



Pointers to pointers are double-cool!

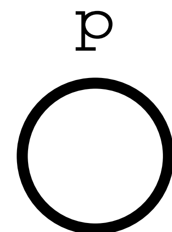
Pointers

```
int x;
 int *p, *q;
 p = &x;
 q = p;
 x = 5;
 *p = 7;
 int **pp;
 pp = &p;
```



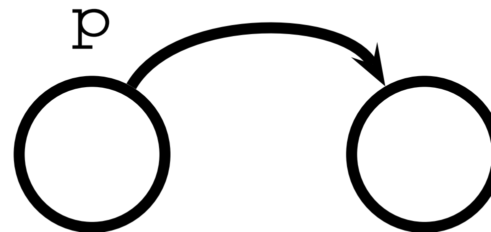
Now `**pp == *p == x == 7`

The `new` operator




```
int *p;
```

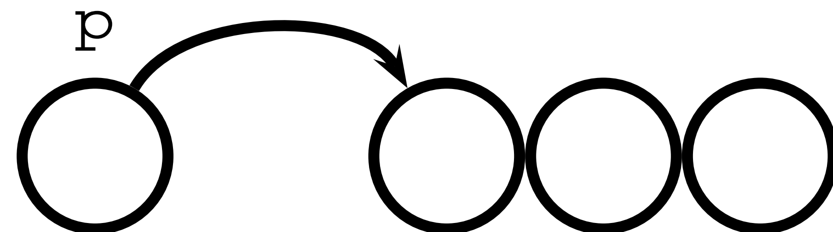
The `new` operator



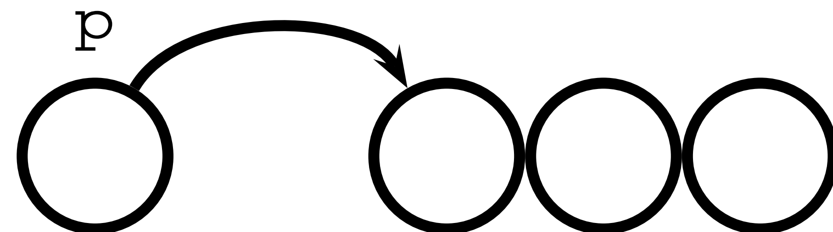

```
int *p;
p = new int;
```

The new operator

```
int *p;
p = new int[3];
```



The new operator

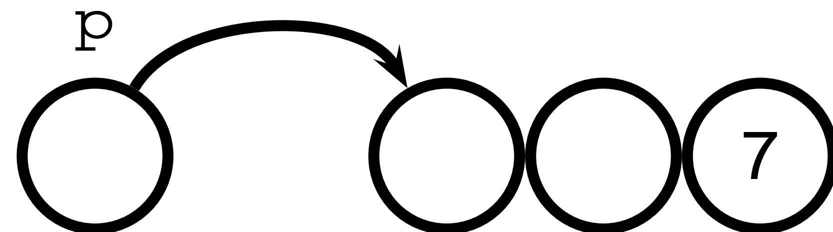



```
int *p;
p = new int[3];
```

Notice that p is pointing to the beginning of the new array

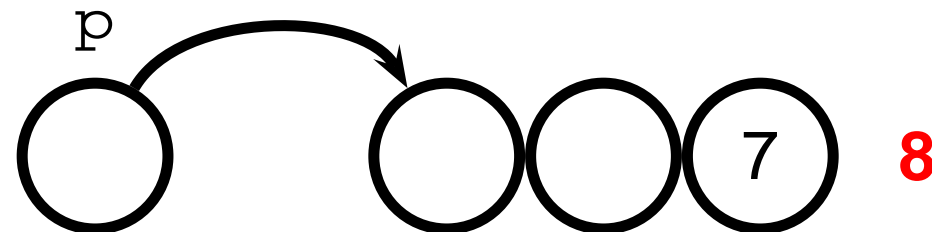
The new operator

```
int *p;
p = new int[3];
*(p + 2) = 7;
```



We can access the rest with pointer arithmetic

The new operator

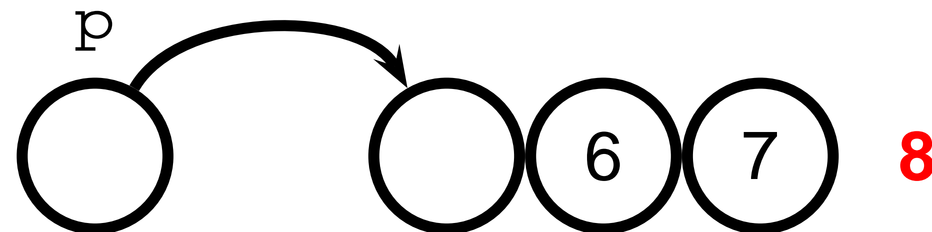



```
int *p;
p = new int[3];
*(p + 2) = 7;
*(p + 3) = 8;
```

But don’t go out of bounds!!

The new operator

```
int *p;
p = new int[3];
*(p + 2) = 7;
*(p + 3) = 8;
p[1] = 6;
```



Pointers can be treated like arrays. `p[i] == *(p+i)`

Pointers as arrays

Pointers and arrays are basically the same thing (syntactic sugar).

```
int a[10];
int *b;
b = a; //Compiler does not complain
b[2] = 5; //a[2] == 5 now as well
*(a+2) = 6; // now both change to 6
```

Arrays and function calls

Array-Pointer equivalence can be confusing when dealing with functions. `void myfunc(int * a);` is the same as `void myfunc(int a[]);`

- The function may change the contents of an array argument (unless `const` is used).
- An array argument is not copied
- The function does not know the array size

Array sizes and functions

```
void myfunc(int a[])
{
        cout << sizeof(a)/sizeof(a[0]) << endl;
}

int main()
{
        int a[10];
        cout << sizeof(a)/sizeof(a[0]) << endl;
        myfunc(a);
}
```

Guess the output of this program…

Call-by-Value

Normally, when you call a function the argument are **copied** into new locations before being passed on to the function.

> **Always** keep this in mind when passing a complex object as argument to a function!

When giving an array as argument however, a pointer to the first element is copied and given to the function.

When a function is given pointers as arguments it may change data values in the main program.

> This is the standard way to do call-by-reference in C.

Swap function in C

```
void swap(int *a, int *b)
{
        int tmp;
        tmp = *a; *a = *b; *b = tmp;
}

int main()
{
        int x,y;
        x = 2; y = 3;
        swap(&x,&y); //Notice the &
}
```

Call by reference in C++

Though you can do the same in C++, a more convenient way is offered: define some function arguments as references.

Pros

- The arguments are not copied (can save space and time and save your from bugs)
- The function can change their values
- No need for annoying *s all over the place

Cons

- As when using pointers, ref arguments must be valid left-hand-side expressions. If `f()` is a function that takes an `int` reference, you cannot say `f(x`3)+. Generally, a left-hand-side expression is anything you are allowed to use to the left of a =.

Swap function in C++

```
void swap(int &a, int &b)
{
        int tmp;
        tmp = a; a = b; b = tmp;
}

int main()
{
        int x,y;
        x = 2; y = 3;
        swap(x,y); //Notice the absence of &
}
```

The `delete` operator

Dynamic memory allocation gives great power. With great power comes great responsibility!

Clean up after yourself!

```
int *x;
x = new int;
...
delete x; //Give memory back when you don't need it

x = new int[10]; //Now x can be reused
...
delete [] x; //Give array back
```

Memory management

Memory leaks

```
int *p;
p = new int[10];
...
p = new int[10];
//Now you cannot access the old array!
//It is lost until your program exits!
```

Memory management

Dangling pointers

```
int *p,*q;
p = new int[10];
q = p;
...
delete [] p;
q[2] = 7;
//That's a No-No!
//This memory is not yours any more
delete [] q;
//That's a No-No!
//You have deleted this already!
```

Guess the output

```
int *p1, *p2;

p1 = new int;
p2 = new int;
*p1 = 100;
*p2 = 200;
cout << *p1 << " , " << *p2 << endl;
delete p1;
p1 = p2;
cout << *p1 << " , " << *p2 << endl;
*p1 = 300;
cout << *p1 << " , " << *p2 << endl;
*p2 = 400;
cout << *p1 << " , " << *p2 << endl;
delete p1;
```